Automated Wallops Orbital Tracking Station (AWOTS)
Design
for
Remote Node Subsystem

June 12, 1995
Version 1.0

NASA
National Aeronautics & Space Administration
Goddard Space Flight Center
Wallops Flight Facility
Wallops Island, Virginia 23337

Table of Contents

List of Figures

1.0 Overview

The Remote Node (Node) subsystem exists in the Automated Wallops Orbital Tracking Station (AWOTS) effort to provide low level control of the station's equipment. The design of AWOTS calls for all non-Ethernet device interaction to be off-loaded from the Master computer to a Remote Node computer. A Remote Node acts a server, to the Master which is the client, providing access to the equipment resources. The communications between the Master and a Remote Node is accomplished with TCP/IP.

The overall requirement of the Node subsystem is to provide control of equipment. This task can be decomposed into the subtasks of equipment reporting, equipment controlling, and equipment monitoring.

A Node must be able to report to the Master the types of equipment available and the quantity of each type (i.e. 5 BrandX recorders). It must also respond to requests from the Master for assignment of equipment given a general description (i.e. a BrandX recorder) or a specific description (i.e. BrandX recorder #2).

Once a piece of equipment is assigned to the Master, the Node must configure and control the equipment. It must provide for scripted control of the device from the Node as directed by the Master, provide low level command control directly from the Master, and provide low level command control from the Node's local user interface.

Finally, the Node must continually monitor the status of the equipment. As directed by a Master the Node will monitor status information specific to each device at a specified time sampling rate. This status information will be communicated back to the requestor. The Node will also monitor the equipment status in order to detect equipment failures. Certain equipment will also be monitored for support events, and the occurrence of such events will be reported and logged.

1.1 Nodes

Multiple Remote Nodes will be employed in the AWOTS. Each Node will control a related subset of the equipment in the station. The Remote Nodes are broken into two groups: those that provide control of non-Ethernet based equipment, and those that integrate new or existing systems. These later nodes are a separate development effort and are not covered here. This document covers the effort of integrating individual pieces of equipment into the AWOTS and the Nodes that control that equipment (see Figure 1-1 on page).

The Nodes covered by this document and the equipment they will be controlling are listed below.

1.1.1 Recorder Node

(2) Datatape 3700J Analog Tape Recorder
(6) Metrum BVLDS Digital Tape Recorder
(6) Metrum BVLDS Serial Control Adapter
(1) Hewlett Packard 3325B Synthesizer Function Generator

(1) Datum 9700 Time Code Generator/Translator
(1) T.B.D. analog matrix
(1) T.B.D. digital matrix

1.1.2 Receiver Node

(5) Microdyne 1620PC Combiner
(1) Hewlett Packard 437B Power Meter
(10) Microdyne 1200MRC Receiver
(1) Hewlett Packard E1301A RF Matrix
(1) Microdyne TSS2000 Signal Generator

1.1.3 Data Handling Node

(2) Matrix Systems, Inc. 10693 Analog Matrix Switch
(12) Decom Systems, Inc. 7715 Bit Synchronizer
(2) NASA Data Generator
(12) General Data Products 225D PCM Frame Synchronizer
(1) General Data Products 233 PCM Simulator
(4) Aydin Monitor 429A PSK Demodulator

1.1.4 Tracking and Command Node

(2) NASA Digital Ranging System
(1) Krohnhite 3905B Dual Tunable Low Pass Filter
(2) Kay 1/4450 S.M. Programmable Attenuator
(5) General Data Products 782 PSK Modulator
(3) NASA Time Capture Unit
(2) NASA Tracking Data Formatter
(2) NASA Transportable Command Processing System

1.1.5 NASCOM Node

(2) General Data Products 545 Block Error Detector
(2) T.B.D. Digital Matrix Switch

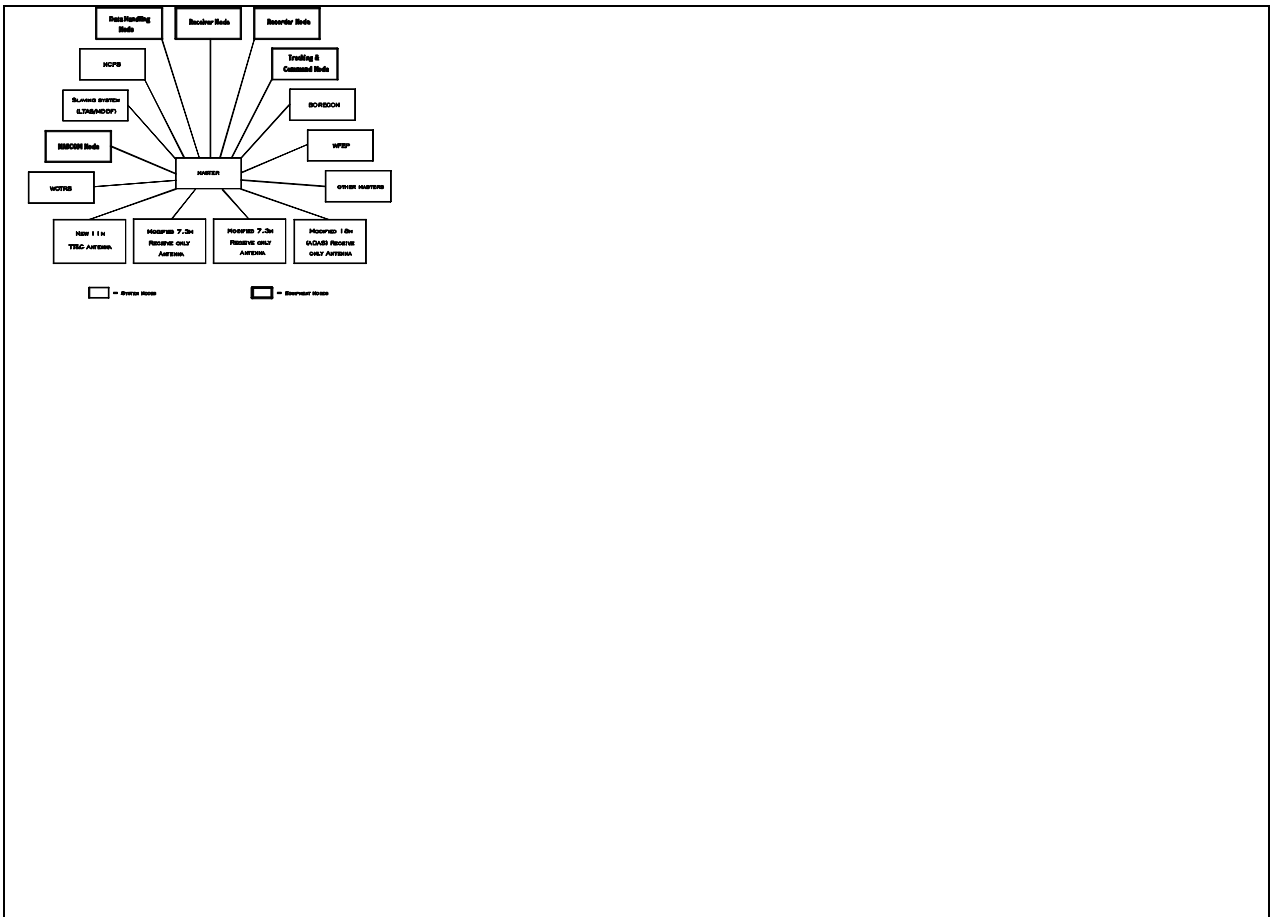

Figure 1-1: AWOTS Systems Overview

1.2 Development Environment

The development environment can be divided into three separate categories: the hardware platform, the operating system, and the development tools. A study was conducted, given the requirements of the AWOTS system, to chose the development environment. The results of this study are available in a separate document, "Wallops Orbital Tracking Station (WOTS) Upgrade Process Software Development and Management Plan". The chosen environment and how it relates to the Remote Node development is detailed below.

1.2.1 Hardware Platform

The chosen hardware platform is an Intel Pentium processor with a combination PCI and ISA bus. The ISA bus allows the use of readily available interface cards, including those already in use at WFF. The PCI bus allows the use of high speed cards for speed critical functions such as network interfacing, hard disk control, and graphics display. The Pentium processor provides the necessary processing speed to effectively run the operating system and provides compatibility with the existing code base.

1.2.2 Operating System

The functional description of a Remote Node necessitates the use of a real-time operating system. The need to simultaneously control multiple pieces of equipment while responding to requests from a Master and possibly a user at the local interface requires a multitasking, time slicing environment. The need to communicate with the Master computers requires support for network communications. The requirement to interface with equipment, including internal ISA cards, requires support for industry standard protocols. All of these factors were taken into consideration when the Microsoft Windows NT operating system was chosen.

1.2.3 Development Tools

Two main tools have been chosen for the development of the software. These are the Microsoft Visual C++ compiler and the BlueWater WinRT toolkit.

Microsoft Visual C++ provides a means to produce source code for the Windows NT operating system. It provides a rich development environment with graphical tools for designing user interfaces and for interactive debugging of source code.

The BlueWater WinRT toolkit provides a solution to interfacing with internal cards. As a level C2 secure operating system, Window NT prevents a program from directly accessing hardware. All hardware access must go through the operating system. This is normally accomplished with device drivers provided by either Microsoft or third party developers. When Windows NT drivers are not available the programmer must write his own, which can be a massive task. The WinRT toolkit provides a high level means of interacting with cards by providing a low level Windows NT driver and a high level programming interface.

1.3 Software Development

There are a number of goals that are desired to be met as part of the Remote Node software development. The primary goal is to produce a working system within the assigned time frame. As a consequence of the primary goal but as a secondary goal unto itself, there is a desire to generate software that can be reused.

The ability to reuse code and the goal to design code to be reused will have benefits not only within the AWOTS project but for future projects as well. Within the project, software reuse can be achieved within a Remote Node, across the Remote Nodes, and across the Master and Remote Nodes.

As much software as determined feasible and desirable will be written in an object oriented method. This will greatly increase the reusability of the code for other projects, possibly under a different operating system and on different hardware. In addition, code will be written to be ANSI standard C and C++ as much as possible. However, it is inevitable that some code must be written that is operating system and/or hardware specific. In these cases the code will be isolated and pushed to the lowest levels so that a minimal amount of rewrite is necessary to achieve reuse and portability. Code will be modularized and will use parameter passing with the use of global values avoided unless absolutely necessary.

1.4 Supporting Documents

Wallops Orbital Tracking Station (WOTS) Upgrade Process, Operations Concept & Requirements for WOTS Master Subsystem and WOTS Remote Nodes Subsystem, Version 1.0, Wallops Flight Facility, Wallops Island, Virginia, January 1994.

Wallops Orbital Tracking Station (WOTS) Upgrade Process Software Development and Management Plan, Version 1.0, Wallops Flight Facility, Wallops Island, Virginia, February 1994

Automated Wallops Orbital Tracking Station (AWOTS) Design for Master Subsystem, Version 1.0, Wallops Flight Facility, Wallops Island, Virginia, June 1995

2.0 Supporting Code

A number of general purpose software components were identified in the Remote Node design. These items were determined to not be AWOTS specific and have been designed to be available for direct reuse on other projects. These items, identified as Supporting Code, have been developed in three main areas: error management, interprocess and intercomputer communications, and equipment interfacing.

A powerful error management capability has been developed. Instead of the traditional use of a single integer value error return from a function call, the caller gains access to a linked list of error information. This information traces an error through the program's execution path to the lowest level where the error actually occurred. The error management scheme provides the line number and function name for each intermediate error and a human readable text description of the error.

Interprocess communications on either the same or different computers has been implemented through the use of system messages modeled on a postal system. A message can be thought of as an envelope with a destination address, a return address, and a letter or other contents. While the contents of the envelopes may be AWOTS specific, the envelopes themselves are not.

Layered device drivers are being developed to interface to all of the equipment controllable by a Remote Node. The layers proceed from command to device to driver to port. While any layer may be directly accessed, the command layer is designed for use by the applications programmer. For example, an external recorder command could be "Record". The device layer would have to issue driver layer commands to see if a tape was loaded and properly formatted before sending it a record command. The port layer would handle any operating system and hardware specific coding.

2.1 Error Management

2.1.1 Overview

When an error occurs in a program it is extremely helpful to know where in the code the error occurred, what the program was attempting to do at the time of the error, and how the program's execution arrived at the code in question. Given the complexity of the Remote Node system and the number of software layers involved, it was determined that a consistent method of error management was needed. The error management system needed to provide for error tracing, error handling, and error reporting.

First, error tracing creates a path from the lowest level program call where an error occurs, through intervening levels, up to a level where the error can be handled. As the error "bubbles up", a list is created of the functions through which program control passes. This provides a powerful tool to aid developers in quickly tracking down the location of errors.

Secondly, error handling allows the program to investigate the type of error that has occurred at the lower levels. The first item in the error list contains the reason for the existence of the error. The other items in the list exist to provide information on the path taken to arrive at the error and what the program was attempting to accomplish when the error occurred. The error codes are enumerated which allows the program to easily determine the type of error and take action.

Finally, error reporting provides a textual description of the error that occurred. The error can be directed to the screen or a file where the error list items are translated from numbers into text that describes for the user or the programmer what has occurred. This is useful for the developer since he can quickly pinpoint the location of an error. It can also be useful to the end user. For example, an error description indicating a problem communicating with a device through a serial port may lead the user to check for a loose cable.

2.1.2 Implementation

It is desirable for all classes to have access to the error management capabilities. This is accomplished through inheritance of these capabilities. In total, three classes are needed to fully support the error management system: the *_Base* class, the *Base* class, and the *Error* class.

2.1.2.1 _Base Class

The _Base class exists solely to provide a common inheritance for the Base class and the Error class (see Figure 2-1 on page). It acts as a repository for the error information of all of the classes in the system. The Base class, through the class Registration mechanism, places the derived class' error information into the _Base class. The Error class provides the means to reference that error information and to retrieve the error text when the errors need to be output (see Figure 2-2 on page).

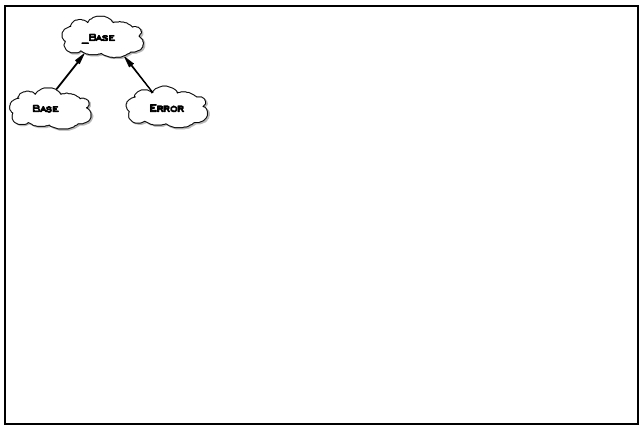

Figure 2-1: Error Management Class Hierarchy

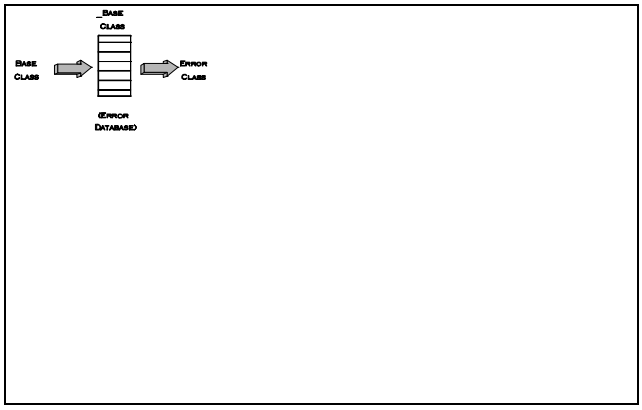

Figure 2-2: Error Management Data Flow

A secondary benefit of this storage scheme is increased reuse. In addition to the code reuse gained by using class inheritance, the error database allows for error message reuse. Just as a child class may make use of a parent class' more general functions, the child class can also make use of a parent class' error messages. This reduces the number of error messages that need to be created at each level, provides reuse, and increases consistency.

2.1.2.2 Base Class

The Base class provides a common inheritance for all other classes. It supplies the Register function which every class must call from its constructor. The Register function places the class' error messages into the database maintained in the _Base class.

2.1.2.3 Error Class

As stated above, the Error class allows the error information stored in the _Base class to be retrieved. However, before the error information can be retrieved and streamed to an output, information about the error that has occurred must be assembled. This is accomplished with the Set member function of the Error class. The Set function adds a link to a dynamic list that contains information about the type of error and the location of the error. As the error message moves up from the lower levels to the higher levels, successive calls to the Set function add new links in the list and create the execution trace path.

When an error finally "bubbles up" to a level where the program is prepared to handle the error, the Error class provides access to the links in the list so that the type of error can be discerned. Finally the error information may be logged to an output device. This is when the database of error information stored in the _Base class is accessed to translate from the internal program representation of the error message to an external human readable representation.

2.2 Interprocess and Intercomputer Communications

2.2.1 Overview

The AWOTS project is built around a real-time, distributed, client-server model. Each of these factors implies that communications between different parts of the system is necessary. This includes communications between processes on a computer (interprocess) and communications between processes on different computers (intercomputer). To ensure that all parts of the system would be able to communicate with each other it was determined that a common messaging system was needed. This messaging system needed to provide common standard features that would allow all processes to understand the purpose of the message but also be adaptable to any data format.

The messaging system is modeled on the standard envelope concept. The outside of the envelope contains the standard destination address and return address. The inside of the envelope contains the item being transported. This is translated into the software world as a message header and a message body.

The message header format is standard for all messages and contains the destination address, the source address, and various other information fields. This standard format allows any message receiver to easily determine the source and intent of the message.

The message body format is user definable. Any type of data can placed in the body. A field in the header indicates to the receiver the type of data included in the body so that the data can be correctly interpreted. The sender and receiver must be aware of the data formats that may be included in the body but beyond that the format is completely open.

2.2.2 Implementation

While the header and the body are the two distinct items in the messaging system model, they are not the classes that are implemented. In the software implementation of the model, the message body becomes more than just the contents but the envelope as well. This translates into the *System Message* class. The header information becomes part of the envelope with the destination and return addresses becoming the second class, the *System Message Address* class (see Figure 2-3 on page).

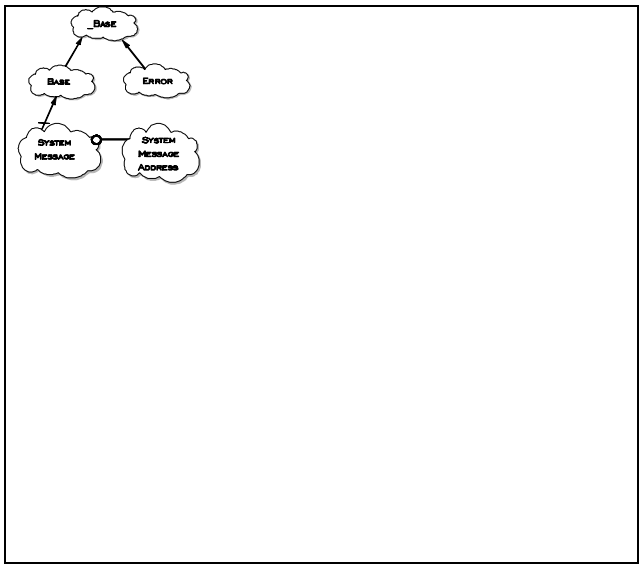

Figure 2-3: System Message Class Hierarchy

2.2.2.1 System Message Address Class

The System Message Address class is used to implement the data structure which contains the information defining a source or destination address. Similar to the common address scheme used by the post office, the address is broken into four fields. The field names have been made generic to facilitate porting to different architectures and to prevent becoming tied to the underlying transport mechanisms. The fields include a Computer Name, a System Name, a Task Id, and a Box Id.

The Computer Name field equates to the state field in the common post office address (i.e. VA for Virginia). This field specifies the computer on which the message originated or to which the message is destined.

The System Name field further pinpoints the source or destination by resolving the address to a specific software system running on the designated computer. The field can be equated to the common post office address field of city (i.e. Wallops Island).

The street name of the common post office address is similar to the Task Id field. This field narrows the address to a specific program, process, or task running within the software system.

Finally, the Box Id field indicates the exact mail slot within the task. The Box Id is a generic name which can represent the name of a software pipe or the number of a software socket.

2.2.2.2 System Message Class

The System Message class implements the envelope and the content metaphor. This is accomplished with the use of the System Message Address class and a dynamic data area.

The information on the outside of an envelope is represented by two instances of the System Message Address class in the message header section. They provide the source and destination address for the message. The header also contains additional fields including:

- a Type field that indicates to the receiver the contents of the message body,
- a Category field that can be used to group the message types,
- an ID field that can be used to track messages and their responses,
- a Certified field that can be used to indicate the desire for an acknowledgement of receipt by the sender,
- a Priority field that can be used by the sender to specify a message processing priority,
- a Key field that can be used for security purposes to authenticate the source of a message,
- and an LSB field to indicate the data byte ordering to facilitate communications between different architectures.

The container property of an envelope is represented by the message body's dynamic data structure. It can hold any type of data and can grow to accommodate any size data item.

Finally, the content of the envelope is exemplified by the data contained within the message body (see Figure 2-4 on page).

The System Message class provides the ability to address the message and to place contents within the message. It provides enough functionality to put data within the message body and retrieve the data from the message body. It does not however have any understanding of the data types placed with the message. The user of the class can work with this limitation, but the task can be made easier if the System Message class is inherited and member functions added to manipulate the user's data types.

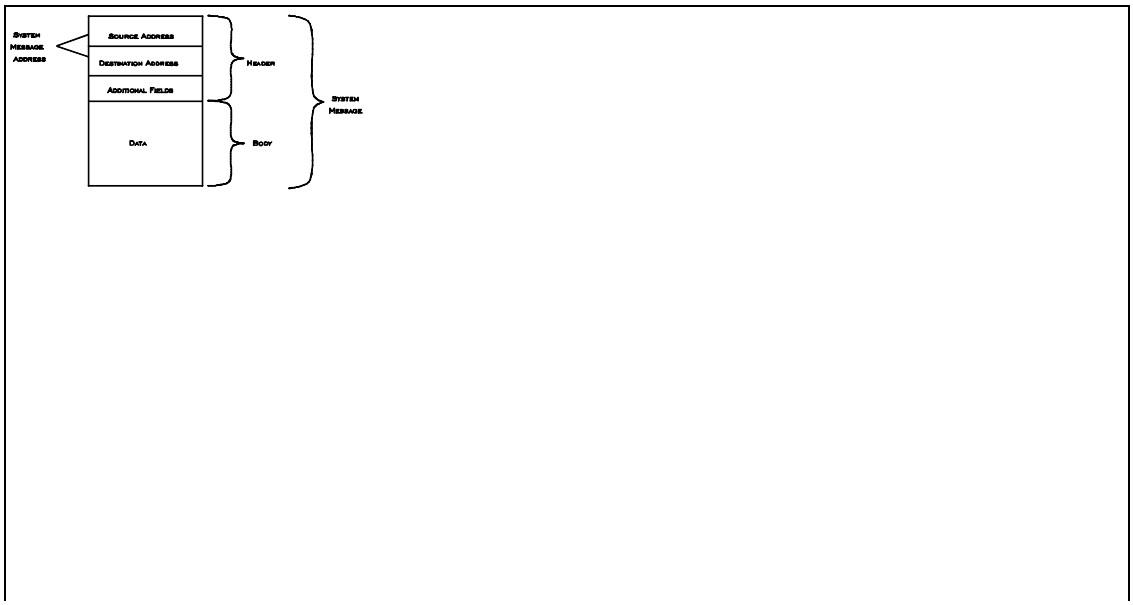

Figure 2-4: Message Format

2.3 Equipment Interfacing

2.3.1 Overview

Interfacing with equipment can be a complex task. This is caused by varying interface protocols, varying equipment command sets, and varying equipment complexities. These factors and the realization that interfacing to the types of equipment in AWOTS was not limited solely to AWOTS but was applicable to other projects as well, impacted greatly on the software design.

The design calls for the implementation of layers of software. Each layer handles a particular task and shields from the other layers the complexities of that task (see Figure 2-5 on page).

The lowest level, Layer 1, interfaces directly with the operating system and/or hardware to implement the interface protocols. Not only does this simplify the higher levels but it provides a "hardware abstraction layer". By placing all of the operating system and hardware interfacing in this layer, the other layers become independent of the operating system and hardware. The other layers can then be easily ported to other systems (different hardware and/or operating system) with only a rewrite of Layer 1 required.

The next level, Layer 2, abstracts the equipment command set. Each command string is wrapped in a function that shields the often cryptic command strings and provides instead a human readable interface. This layer communicates with Layer 1 to facilitate the sending and receiving of data to and from the equipment.

Layer 3 addresses the equipment complexities. While some equipment may be less complex and need no more than the sending of the commands and the receiving of the responses, other equipment can have detailed steps that must be followed for the correct commanding of the device. These complexities are hidden at this level. The equipment user interacts with the functions at this layer. This layer then interacts only with Layer 2 to implement its functionality. Each function at Layer 3 may make calls to one or more functions at Layer 2.

Finally, Layer 4 deals with the need to remotely issue commands to a piece of equipment from a computer to which to the device is not connected. Normally a program would interface with Layer 3 by performing local calls to the functions at that level. However, a remote computer cannot make local calls to the functions in this layer which exist on another computer. The solution is to provide a means to send commands to Layer 3 that cause the level's associated functions to be executed, in effect causing a remote function call. Layer 4 deals with the formation of these commands and the interpretation of the results. The function calls in Layer 4 mirror as closely as possible the calls in Layer 3 thus providing the user with a consistent interface.

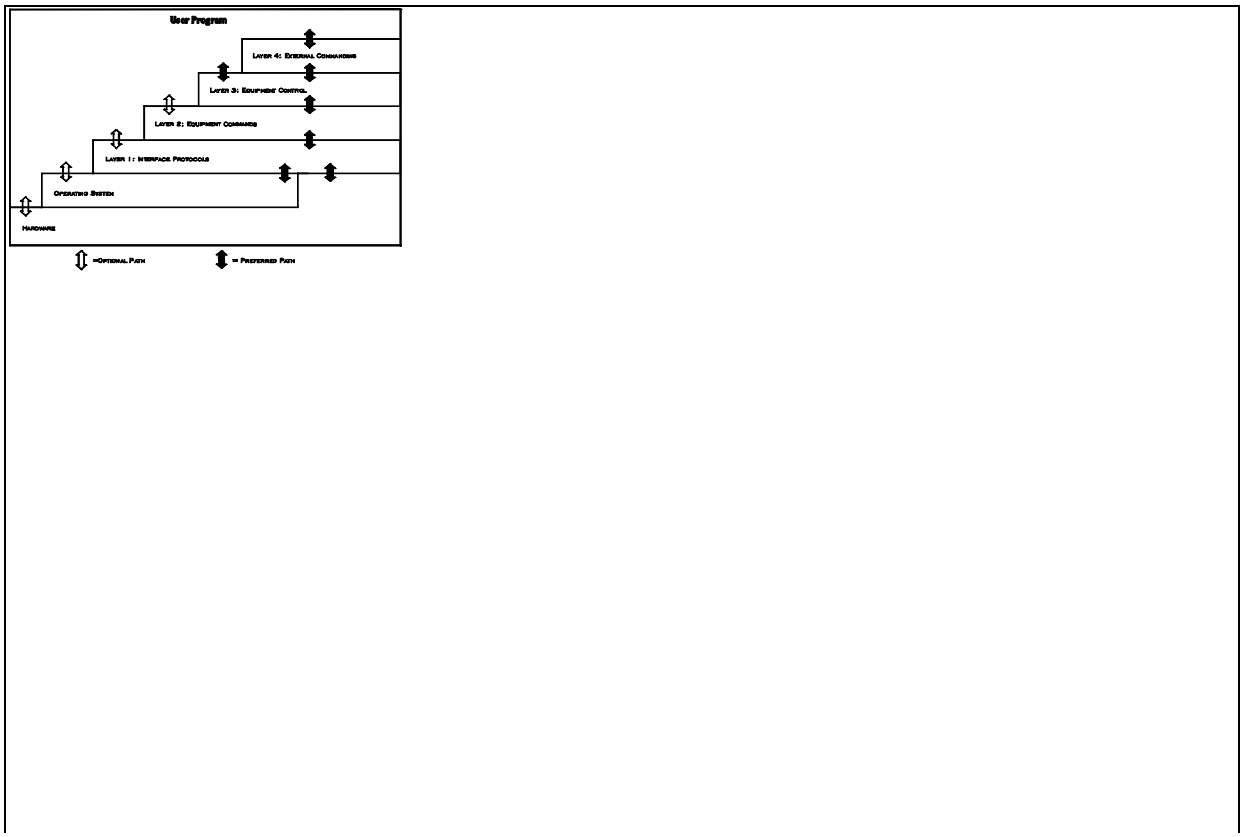

Figure 2-5: Equipment Interfacing Software Layers

2.3.2 Implementation

Each layer in the above design translates directly into a class. Layer 1, the hardware abstraction layer, becomes the *Port* class. The command set abstraction layer, Layer 2, becomes the *Driver* class. The layer that hides the equipment complexities, Layer 3, becomes the *Device* class. And the final layer, Layer 4, that implements remote commanding becomes the *Command* class (see Figure 2-5).

Each layer is actual a family of classes. The Port, Driver, Device, and Command classes form the base classes of each family. Additional classes are derived from each base providing more detailed implementations for specific interface protocols or makes of equipment (see Figure 2-6 on page).

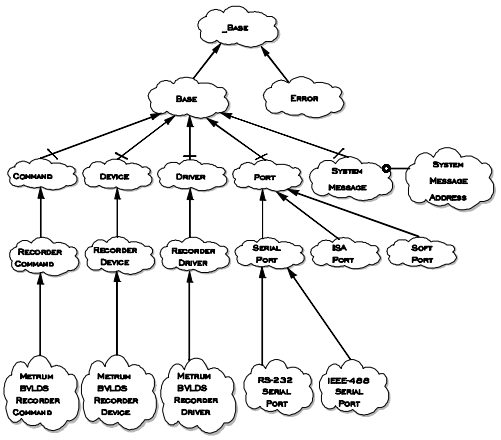

Figure 2-6: Equipment Interfacing Class Hierarchy

2.3.2.1 Port Class

As stated above, the Port class exists to shield the other code from the operating system and hardware interfacing. An abstract base class called Port exists to provide a common inheritance for all types of port classes. From this Port class, another abstract class Serial is derived to provide a common inheritance for serial ports. Finally, from the Serial class the RS-232 and IEEE-488 classes are derived.

Other ports that are derived from the base Port class include the ISA port and the Soft port. The ISA port provides a common base to interface with ISA bus cards. The Soft port is not a physical port but a virtual port. It does its input and output to files and provides a means to debug the device and driver classes without having the equipment connected and to perform data playbacks and simulations. This can be accomplished because of the object oriented feature of polymorphism. This allows a driver to communicate with a port class that has been derived from the base Port class. This feature not only aids in debugging by allowing the Soft port to stand in for the equipment but allows for the development of drivers for equipment that can work with multiple protocols.

2.3.2.2 Driver Class

The Driver class implements the equipment command abstraction. Equipment command formats vary greatly from sending a command number, to sending cryptic command strings, to sending semi-English command strings.

The design calls for the Driver class to remain uncomplicated. Its sole purpose is to provide a higher level interface to the equipment commands. The majority of driver functions are one line calls to a port class supplying the correct command strings. Some equipment commands require parameters that must be supplied to the driver by the user. Although the parameters should be checked for proper range values, to minimize the complexity of the driver level this logic is left to the caller (see the Device class below).

Like the Port class, discussed above, the Driver class makes use of inheritance. An abstract base class called Driver exists to provide a common inheritance for all types of driver classes. From the Driver class an intermediate layer of classes is derived to provide a common inheritance for generic equipment types. For example, the Recorder Driver class is derived from Driver and exists to provide a common inheritance for all types of recorder equipment drivers. Finally, the last derivation layer implements specific makes and models of equipment. The Metrum BVLDS Recorder Driver class, for example, implements the driver for the Metrum make and BVLDS model of digital recorder.

2.3.2.3 Device Class

Equipment control complexities are hidden in the Device class. The Device class provides a high level of equipment control and shields the user from the sometimes complex sequences of equipment commands that must be issued to accomplish a single task.

The design calls for the Device class to implement all of the logic necessary to properly control a piece of equipment. This includes screening user input for proper values, implementing correct command sequences, and interpreting equipment responses. To implement the equipment control, the Device class makes one or more calls to the associated Driver class.

Like the Driver class, the Device class acts as an abstract base class to provide a common inheritance for all types of device classes. From the Device class an intermediate layer of classes is derived to provide a common inheritance for generic equipment types. The Recorder Device class, for example, is derived from the Device class and exits to provide a common inheritance for all types of recorder equipment devices. Finally, the last derivation layer implements specific makes and models of equipment. The Metrum BVLDS Recorder Device class, for example, implements the device for the Metrum make and BVLDS model of digital recorder.

2.3.2.4 Command Class

The Command class satisfies the need to remotely control a device. The Command class provides an interface that echoes the interface to the device class for a particular piece of equipment. Thus code developed to control a piece of equipment locally on the current computer could also control the same piece of equipment on a remote control through the substitution of the command class for the device class.

The Command class formats a string containing an indicator of the actual device function to be executed and any related parameters. The command string is then sent to the computer to which the actual device is connected. The Device class for the equipment on the remote computer parses the command string and executes the desired function. The results of the function call are formatted into a response string which is sent back to the originating computer. The Command class then parses the response string and returns the values to the caller.

Like the Device class, the Command class acts as an abstract base class to provide a common inheritance for all types of command classes. From the Command class an intermediate layer of classes is derived to provide a common inheritance for generic equipment types. The Recorder Command class, for example, is derived from the Command class and exits to provide a common inheritance for all types of recorder equipment command classes. Finally, the last derivation layer implements specific makes and models of equipment. The Metrum BVLDS Recorder Command class, for example, implements the command class for the Metrum make and BVLDS model of digital recorder.

3.0 Node Processes

A Remote Node has five distinct types of processes. The processes each perform an individual task but working together satisfy the requirements of the Remote Node. Node processes will be developed for device control, resource management, message logging, message processing, and user interaction (see Figure 3-1 on page). The processes make use of the supporting code described in section 2.0.

A separate Device Controller process will be provided for each device on the Node. A Device Controller process receives system messages containing device commands. The embedded command is extracted and passed to the device using the layered interfacing techniques described in Section 2.3. When the device responds, the results are packaged in a system message and sent back to the requestor via the return address of the system message.

The Wallops Resource Manager process has the responsibility for coordinating access to the devices on a Remote Node. A device registry file on each Node will contain a list of attached devices and their port connections. The Resource Manager will read this file to determine what devices are available and which drivers and ports to use. It will then service requests from callers to open and close the devices.

A Message Logger process will log system messages to an I/O stream such as a file. More than one Message Logger process may be run at a time. For example, one Message Logger may log all commands received from a Master, and another may log only error messages.

The Message Processor process will receive system messages, typically from a Master, and handle them. The Message Processor will have two pipes for incoming messages, one with a high priority, and one with low priority. The high priority pipe will handle commands, while the low priority pipe will handle status requests.

The User Interface process handles mouse and keyboard input and displays information to the user. The User Interface to each device is identical to the device interface on the Master.

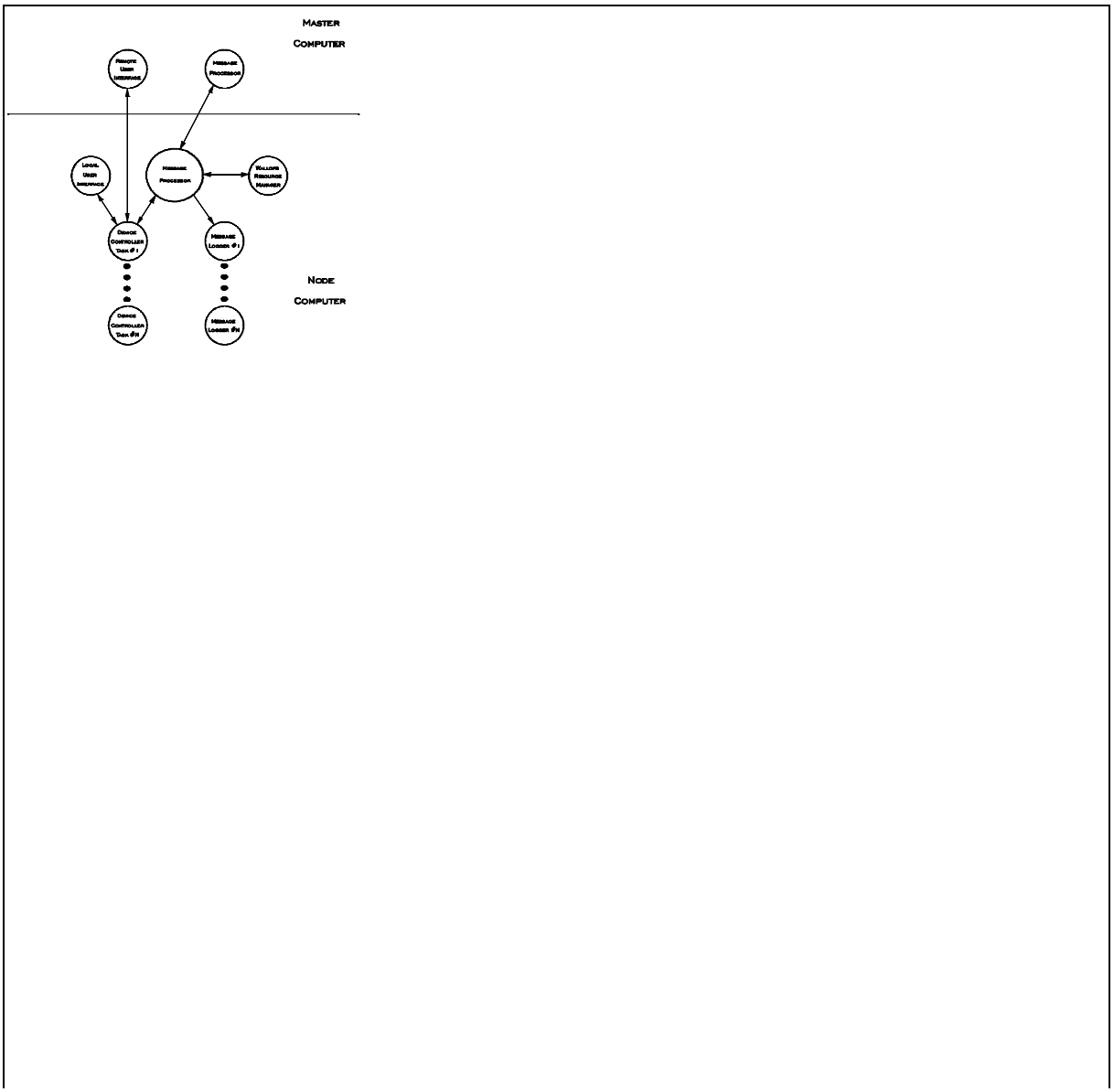

Figure 3-1: Node Processes

3.1 Device Controller Process

3.1.1 Overview

In the overall AWOTS scheme a Remote Node acts as a server to the Master client. A Node provides access to the equipment resources of the station. As if looking at a fractal, closer examination of a Node reveals a similar client/server architecture within the Node. Here, each individual piece of equipment is controlled by a Device Controller which provides access to that piece of equipment. A Device Controller handles requests for equipment services and directs responses from the equipment back to the requestor.

A majority of the service requests to a Device Controller will come from the Node. This will be in response to the Node executing a high level command initiated by the Master. However the Master is also capable of issuing direct equipment control. In this case the equipment command comes directly from the Master and the response if directed back to the requestor bypassing the higher level Node processing altogether.

A Device Controller Process is not AWOTS specific and may be reused, however it does contain some operating specific code.

3.1.2 Implementation

A Device Controller process is created by the Wallops Resource Manager (see section 3.2) when a request is made to open a device. The Resource Manager spawns the process and supplies it with a pointer to an object of the Device class (see section 2.3) corresponding to the specific piece of equipment to be controlled. For each individual piece of equipment there is a corresponding Device Controller process and each Device Controller controls only a single piece of equipment.

Each piece of equipment is classified as belonging to one of four categories: a Write Only device (W) that only accepts input, a Read Only device (R) that only provides output, a Write then Read device (WR) that responds to commands, and a Read or Write (RW) device that responds to commands but may generate spontaneous feedback. A Device Controller process will exist for each of the device categories and will be tailored for its characteristics. When a device is opened for use the correct Device Controller will be created.

The Device Controller process is not specific to any device. The coding of the process is generic and the process can control any of the devices that fall within its category, which allows for process level reuse. This is accomplished by exploiting the object oriented feature of polymorphism. By having the Device Controller task manage a device at the Device class level, any class derived from the Device class can be substituted. Through the use of a few virtual functions any device can be controlled without specific knowledge of the type of device. These virtual functions are Execute, Read, and Unsolicited Response Handler.

All of the four device types will need at a minimum the Execute function. The Execute function provides a means to send a command to the Device object to be carried out. Since the Execute function exists at the Device class base level, all derived device classes support this function. The Execute function must accept a command string containing an enumerated value corresponding to the desired function to be executed and the values of any necessary parameters. Once it calls the desired device function it must create a response string, again containing an indication of the function and the values of any parameters to be returned. A thread of execution is created in the Execute function for each command received by the Device Controller process.

For the WR and RW device types it is necessary to monitor the device for responses. This is accomplished with the Read thread. Started upon the opening of a device, the Read thread receives information from the device, parses it to determine what the data represents, and then stores the information in a table and signals the arrival of the information. Any threads that may have been executed and waiting for the data from the device will become aware of its arrival and continue processing.

Finally, the last Device level function is the Unsolicited Response Handler (URH). Like the Read thread it is immediately started for those devices that provide feedback. It blocks on entries in the data table that represent anomalous responses. It can then determine how to handle the conditions based upon the particular device. If necessary it can send a message to another process.

A Device Controller process for a RW device is shown below. The Execute, Read, and Unsolicited Response Handler are shown interacting with the Device object. The Device object's communication with the equipment is shown with the Driver object and the Port object. The Command thread is a thread that is provided by the Device Controller. It retrieves messages from the incoming pipe and acts upon each. If the message is for the device the Command thread passes the message to an Execute thread. If the message is for the Device Controller (a shutdown message for instance) then it is handled outside of the device object.

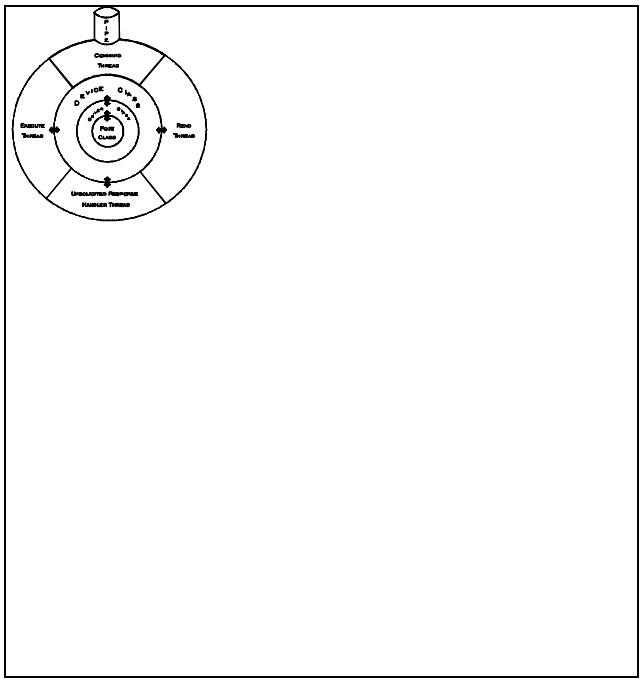

Figure 3-2: Device Controller Process

3.2 Wallops Resource Manager Process

3.2.1 Overview

The Wallops Resource Manager (WRM) manages the use of the equipment resources by other processes. It determines what resources are attached to the computer, loads the appropriate device drivers, and responds to requests for access to the equipment.

The Wallops Resource Manager is not AWOTS specific and may be reused, however it does contain some operating specific code.

3.2.2 Implementation

The WRM determines the resources attached to the computer and the method of attachment by reading the Wallops Resource Manager Registration file. The file contains three sections: a Ports section, a Devices section, and a Connections section. The Ports section contains an entry for each of the available interfaces on the computer such as RS-232 ports, IEEE-488 ports, RS-232 multiplexers, and IEEE-488 converters. Each entry contains a port type and a port number. The Devices section lists each individual piece of equipment attached to a port on the computer. Each entry contains a device type and a unit number. Finally, the Connections section shows how the devices are connected to the ports. A connection may be as simple as a device on a port or may contain a device connected to a chain of linked ports.

Example of WRM Registration File:

```
[Ports]
\RS232\0
\RS232\1
\IEEEConverter\0
\IEEEConverter\5

[Devices]
\MetrumBVLDS\1
\Microdyne1620PC\1

[Connections]
\RS232\0\MetrumBVLDS\1
\RS232\1\IEEEConverter\5\Microdyne1620PC\1
```

Once the registration file has been read, the WRM instantiates the correct Port, Driver, and Device objects (see Section 2.3) to control the equipment. It is then ready to respond to messages from other processes running on the computer.

The messages that the WRM can responds to are: Query, Open, and Close. Query returns a list to the requestor of the available equipment. The Open message causes the WRM to start a

Device Controller process to control a designated piece equipment (see Section 3.1) and returns a handle which can be used by the requestor to communicate directly with the controller process. Depending upon the resource, a piece of equipment may be opened simultaneously by multiple processes. Finally, the Close message releases the resource from being allocated to a process. When all processes have closed a resource the associated Device Controller process is killed.

3.3 Message Logger Process

3.3.1 Overview

The Message Logger process logs system messages to an I/O stream such as a log file. Multiple copies of the Message Logger process may be run, each with a different I/O stream. In addition, multiple processes and threads may log messages to the same stream.

The Message Logger Process is not AWOTS specific and may be reused, however it does contain some operating specific code.

3.3.2 Implementation

A Message Logger process will be controlled by the Wallops Resource Manager. Prior to logging any messages, a call must be made to the Wallops Resource Manager to open the message log. When the first request to open a message log to an I/O stream is received, the I/O stream will be opened, a Message Logger process will be spawned, and a handle to its named pipe will be passed to the caller. Subsequent calls by other processes or threads to open a message log on the same I/O stream will also receive its pipe handle. A count will be maintained of the number of processes or threads which have opened the message log. As calls to close the log are received, the count will be decremented. When a close call is received which brings the open count to zero, the handle will be deleted, the I/O stream closed, and the Message Logger process terminated.

3.4 Message Processor Process

3.4.1 Overview

The most AWOTS specific process in the Remote Node subsystem will be the Message Processor process. As such it may not be directly reused, however it can be used as a model for creation of similar processes on other projects. As its name implies, it will process system messages received from the Master or other processes on the Node.

The Message Processor will have a high priority pipe and a low priority pipe. Messages on the high priority pipe will be serviced before those on the low priority pipe. High priority messages will consists of high level commands from the Master such as "Setup BrandX recorder #1 for support". Messages on the low priority pipe will be serviced after those on the high priority pipe and will consist of status requests such as "Report status for BrandY bit sync every 5 seconds".

3.4.2 Implementation

The Message Processor will consist of a high priority pipe server and a low priority pipe server. When a message is received on one of the pipes a thread will be created to handle it. A new thread will be created to handle each message.

The thread will determine how to handle the message by examining the system message category and type fields (see Section 2.2). If the category or type indicates that the message can not be handled by this Node then the message will be returned to the sender. If the message is acceptable then the appropriate functions will be called to handle it.

The AWOTS Remote Node Message Processor will be able to handle messages for any of the equipment in the AWOTS station. The functions to handle the messages will be dynamically loaded at the startup of the Message Processor based upon the type of equipment reported to be available by the Wallops Resource Manager (see Section 3.2). This will enable one set of Remote Node software to be installed on each Node thus making the Nodes interchangeable and easily replaceable in the event of a hardware failure.

The high level equipment control functions will be coded based upon input from the operators and engineers. The functions will interact with the Device Controllers (see Section 3.1) to correctly configure and monitor the equipment. Status and event information will be gathered and sent to log files (see Section 3.3)

3.5 User Interface Process

3.5.1 Overview

The User Interface on the Remote Node system exists to provide the AWOTS user with a fall back position in the event that the Master computer or the network system fails. The interface screens that appear on a Remote Node are a subset of those available on the Master and will be determined by the type of equipment attached to a particular Node.

The User Interface will allow the user to monitor the status of the Node and directly control the equipment attached to that Node. It will not support control of equipment attached to another Node and thus will not support control of the entire station.

3.5.2 Implementation

The Remote Node system will reuse the User Interface processes developed for each device on the Master system. Since interprocess communication is carried out through system messages which include a return address, the location, either local or remote, of the active User Interface will be transparent to the Message Processor and the Device Controller processes (see Figure 3-1 on page). For detailed information on the User Interface process refer to *"Automated Wallops Orbital Tracking Station (AWOTS) Design for Master Subsystem"*.

Appendix A: Abbreviations and Acronyms

AWOTS	Automated Wallops Orbital Tracking Station
ISA	Industry Standard Architecture
NASCOM	NASA Communications
PCI	Peripheral Component Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
TT&C	Telemetry Tracking and Command
WRM	Wallops Resource Manager